PROCEEDINGS OF SCIENCE

Staggered fermions simulations on GPUs

Claudio Bonati*
Dipartimento di Fisica, Università di Pisa and INFN, Largo Pontecorvo 3, I-56127 Pisa, Italy.
E-mail: claudio.bonati@pi.infn.it

Guido Cossu
KEK Theory Center, 1-1 Oho, Tsukuba-shi, Ibaraki 305-0801, Japan.
E-mail: cossu@post.kek.jp

Massimo D'Elia
Dipartimento di Fisica, Università di Genova and INFN, Via Dodecaneso 33, 16146 Genova, Italy.
E-mail: massimo.delia@ge.infn.it

Adriano Di Giacomo
Dipartimento di Fisica, Università di Pisa and INFN, Largo Pontecorvo 3, I-56127 Pisa, Italy.
E-mail: digiaco@df.unipi.it

We present our implementation of the RHMC algorithm for staggered fermions on Graphics Processing Units using the NVIDIA CUDA programming language. While previous studies exclusively deal with the Dirac matrix inversion problem, our code performs the complete MD trajectory on the GPU. After pointing out the main bottlenecks and how to circumvent them, we discuss the performance of our code.

The XXVIII International Symposium on Lattice Field Theory
June 14-19,2010
Villasimius, Sardinia Italy

*Speaker.

1. Introduction

In recent years the video game market developments compelled graphic processing units (GPUs) manufacturers to increase the floating point calculation performance of their products, by far exceeding the performance of standard CPUs. The architecture evolved toward programmable many-core chips that are designed to process in parallel massive amounts of data. These developments suggested the possibility of using GPUs in the field of high-performance computing as low-cost substitutes of more traditional CPU-based architectures.

The introduction of GPUs in lattice QCD calculations started with the seminal work of Ref. [1], in which the native graphics APIs were used, but the real explosion of interest in the field followed the introduction of NVIDIA's CUDA (Compute Unified Device Architecture) platform, that effectively disclosed the field of GPGPU (General Purpose GPU [2]).

Previous studies on the application of GPUs to lattice QCD calculations were mainly aimed at using them together with the standard architectures in order to speed up some specific steps, typically the expensive Dirac matrix inversion. Our intent is to use GPUs in substitution of the usual architectures, actually performing the whole simulation by them. To achieve this result we found simpler to write a complete program from scratch instead of using existing software packages[1], in order to have a better control of all the steps to be performed and ultimately transferred to the GPU. Our implementation uses NVIDIA's CUDA platform together with a standard C++ serial control program running on CPU.

2. The algorithm

To simulate N_f flavours of staggered fermions the Rational Hybrid Monte Carlo (RHMC) algorithm, introduced in [3], has become the standard choice. We used standard (*i.e.* non-improved) staggered fermions and, to speed-up the simulations, the following common tricks were implemented

- even/odd preconditioning
- multi-step integrator (action divided in gauge and fermion part)
- improved integrator (second order minimum norm)
- multiple pseudo-fermions to reduce the fermion force magnitude and increase integration step size
- different rational approximations and stopping residuals for Metropolis step and Molecular Dynamic (MD)

3. A GPU scratch

NVIDIA GPUs are massively parallel computing elements, composed of hundreds of cores (called streaming processors) grouped into multiprocessors. The typical architecture of a modern NVIDIA graphic card is outlined in Fig. 1.

[1] On earlier stage we wrote a staggered version of JLab's Chroma working on GPUs.

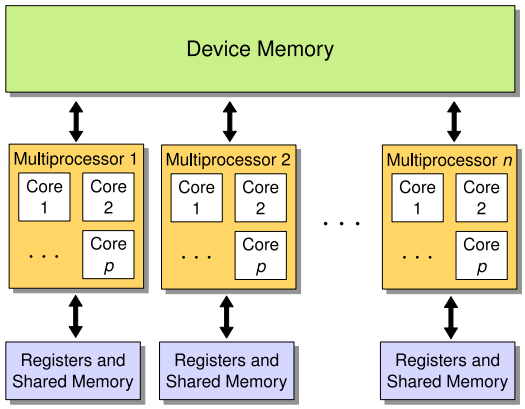

Figure 1: Architecture of a modern NVIDIA graphics card.

Three different storage levels are present: primary storage is provided by the device memory, which is accessible by all multiprocessors but has a relatively high latency. Within the same multiprocessor, cores have access to local registers and to shared memory, which is shared between the threads of the multiprocessor and it is orders of magnitude faster than device memory, being very close to the computing units. While the total amount of device memory is of order of GBs, the local storage is only 16KB both for the registers and for the shared memory[2], so that it is typically impossible to use just these local fast memories. The latency time of the device memory can be hidden by having a large number of threads in concurrent execution, so when data are needed from device memory for some threads, the ones ready to execute are immediately sent to computation. The highest bandwidth from device memory is achieved when a group of 16 threads accesses a contiguous memory region (coalesced memory access), because its execution requires just one instruction call, saving a lot of clock-cycles. This will be crucial in the following, when discussing the storage model for the gauge configuration.

Double precision capability was introduced with NVIDIA's GT200 generation, the first one specifically designed having in mind HPC market, and by now there is only a factor 2 between the peak performance in single and double precision. In Tab. 1 the specifications of the GPUs used in this work are reported.

Communications between the GPU and the CPU host are settled by a PCI express bus, whose typical bandwidth is 5GB/s, to be compared with the GPU internal bandwidth between device memory and cores of order 100 GB/s. This is clearly the main bottleneck in most of GPU ap-

GPU	Cores	Bandwidth GB/s	Gflops (peak) single	Gflops (peak) double	Device Memory GB
Tesla C1060	240	102	933	78	4
Tesla C2050/2070	448	144	1030	515	3/6

Table 1: Specifications of the NVIDIA cards used in this work.

[2]For NVIDIA Tesla cards 10 series. The 20 series has 64KB of on-chip memory that can be partitioned as shared and L1 cache.

$b_{11}(1)$	$b_{11}(2)$	$b_{11}(3)$	$\cdots$	$\cdots$	$b_{12}(1)$	$b_{12}(2)$	$b_{12}(3)$	$\cdots$	$\cdots$
$\cdots$	$b_{22}(1)$	$b_{22}(2)$	$b_{22}(3)$	$\cdots$	$\cdots$	$b_{23}(1)$	$b_{23}(2)$	$b_{23}(3)$	$\cdots$
$c_{11}(1)$	$c_{11}(2)$	$c_{11}(3)$	$\cdots$	$\cdots$	$c_{12}(1)$	$c_{12}(2)$	$c_{12}(3)$	$\cdots$	$\cdots$
$\cdots$	$c_{22}(1)$	$c_{22}(2)$	$c_{22}(3)$	$\cdots$	$\cdots$	$c_{23}(1)$	$c_{23}(2)$	$c_{23}(3)$	$\cdots$

Figure 2: Gauge field storage model: the element $u_{ij}(k)$ of the link k is given by $u_{ij}(k) = b_{ij}(k) + c_{ij}(k)$. b_{ij} and c_{ij} are respectively the most and the less significant bits of u_{ij}.

plications. We thus decided to copy the starting gauge configuration (and momenta) on the device memory at the beginning of the simulation and to perform the complete update on the GPU, instead of using it just to speed up some functions and transferring gauge field back and forth between host and device memories.

In our implementation of the Dirac kernel a different thread is associated to every (even) site in the fermion update and to every link in the gauge update, so that different treads do not cooperate. Shared memory is thus used just as a local fast memory. This setup is forced by the high ratio between data and floating point operations per kernel.

4. Precision issues

We will now address the issues related to the use of double precision. The main drawback of double precision is clear from Tab. 1: single precision floating point arithmetic always outperforms the double one, although in the Fermi architecture the double precision penalty was significantly reduced. Another motivation to prefer the single precision is to speed up memory transfers because lattice QCD calculations are typically bandwidth limited.

Nevertheless double precision appeared to be necessary in the evaluation of the action for the Metropolis step to be performed at the end of a MD trajectory, which guarantees the correctness of the RHMC algorithm (see also Sec. 6). Because of that the first and the last Dirac inversions are performed in double precision, while the inversions needed in the fermion force calculation are in single precision. The update of the gauge field is performed in single precision and double precision is used only in the reunitarization.

Another important feature that is necessary for the algorithm to be exact is the reversibility of the trajectories [4]. Since the gauge updates use only single precision we can expect reversibility to be valid only up to single precision and this is indeed the case: the magnitude of the reversibility violation for degree of freedom is measured to be of order 5×10^{-6}.

5. Memory allocation scheme

We noted previously that a correct allocation scheme is of the utmost importance in order to efficiently use the device memory. For staggered fermions, the storage of the gauge configuration is the most expensive one, so we will concentrate on this. Similar techniques can be used also for the momenta and the pseudo-fermions storage.

Lattice	Bandwidth GB/s	Gflops
4×16^3	56.84 ± 0.03	49.31 ± 0.02
32×32^3	64.091 ± 0.002	55.597 ± 0.002
4×48^3	69.94 ± 0.02	60.67 ± 0.02

Table 2: Staggered Dirac operator kernel performance figures on a C1060 card (single precision).

As stated before, QCD calculations on GPU are typically bandwidth limited and so it is convenient not to storage all the elements of an $SU(3)$ matrix (18 real numbers), but to use a representation in terms of fewer parameters. In this way we can reduce the amount of memory exchange at the expense of increasing the computational complexity. The additional calculations do not introduce significant overhead, actually they are negligible compared to the memory transfers. We used a 12 real number representation: only the first two rows are stored, while the third is reconstructed on fly.

Since in the Metropolis step the inversion of the Dirac matrix in double precision is required, we need to store a double precision gauge configuration, although in most of the calculations it will be used just as a single precision one. In order not to waste bandwidth and device memory, it is useful to write a double precision number a by using two single precision numbers b and c: b is defined by the 32 most significant bits of a, while c stores the 32 less significant ones. In C language this amounts to

$$b = (\texttt{float})a$$
$$c = (\texttt{float})(a - (\texttt{double})b)$$

When only single precision is required we can just use b instead of a, otherwise we have two possible choices: to use b and c directly, effectively avoiding the explicit use of double precision arithmetic (see *e.g.* [5]), or to reconstruct the double precision number a to be used in calculations. We implemented this last method, which is expected to be more efficient on double precision capable hardware.

To get coalesced memory accesses it is crucial for blocks of thread in execution to use contiguous regions of device memory. This behaviour is maximized if we adopt the storage model shown in Fig. 2; the use of texture memory is a further improvement to reduce the effects of imperfect memory accesses.

The performance of the Dirac operator in single precision which is obtained by means of this storage scheme is shown in Tab. 2. From these data it is clear that the main bottleneck is the bandwidth: while using $60 - 70\%$ of the bandwidth, only the $5 - 6\%$ of the peak performance is reached.

6. Inverter

The inversion of the Dirac operator in lattice QCD simulations is usually performed by using Krylov space solvers; for staggered fermions the optimal choice is the simplest one of this class of solvers: the Conjugate Gradient (CG) algorithm.

In all Krylov space solvers the approximate solution and its estimated residual are calculated recursively. While in exact arithmetic the estimated residual is exactly the residual of the approximate solution, in finite precision this is no more the case. The attainable accuracy at fixed precision can be estimated [6] and for a single precision Dirac inversion a typical value for the minimum true residual is $10^{-2} - 10^{-3}$, so we need double precision to perform the inversions related to the Metropolis step.

In standard Krylov solvers this problem can be overcome by using the residual replacement strategy: sometimes the true residual is explicitly calculated in double precision and the algorithm is restarted. With this method it is possible to obtain reliable results, as with double precision calculations, but using almost always single precision arithmetic. Residual replacement methods are well understood theoretically [7] and have been successfully applied to QCD calculations on GPUs [8]. However, in RHMC we need Krylov solvers for shifted systems (see *e.g.* [9]), whose starting solution has to be the null one, thus preventing the restarting of the algorithm. For this reason the Dirac inversions in the Metropolis step have to be performed completely in double precision.

7. Performance

In Fig. 3 the RHMC update time on different architectures is shown for two values of the bare quark masses ($am = 0.01335, 1.0$). For both the mass values the scaling with the size of the lattice is good. In fact it is a characteristic feature of GPUs that increasing the lattice size improves the computational efficiency, as seen also in Tab. 2; this happens because with large lattices internal latencies are hidden more effectively. Time gains for Tesla C1060 and C2050 are shown in Tab. 3 and Tab. 4; particularly impressive is the comparison with the results obtained by using an apeNEXT crate.

8. Conclusions

The extremely high computation capabilities of modern GPUs make them attractive platforms for high-performance computations. Previous studies on lattice QCD applications have been de-

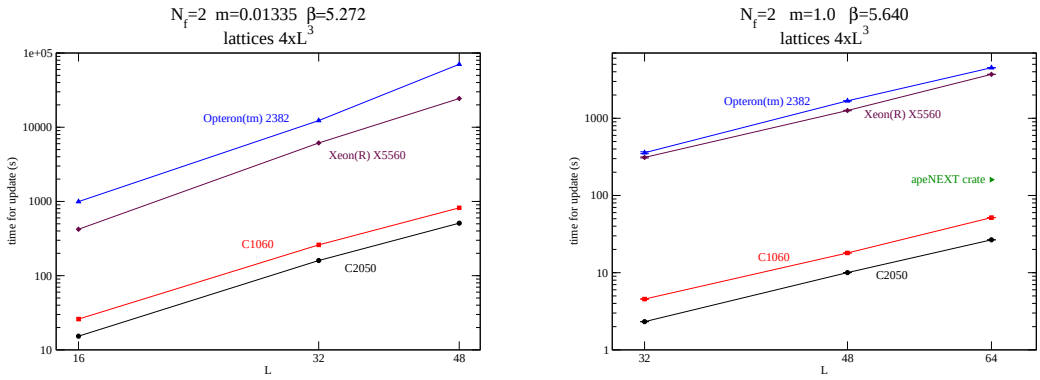

Figure 3: Run times on different architectures. For the Opteron and Xeon runs a single core was used.

	high mass			low mass		
spatial size	32	48	64	16	32	48
Opteron (single core)	65	75	75	40	50	85
Xeon (single core)	50	50	50	15	25	30
apeNEXT crate	$\sim$3			$\sim$1		

Table 3: NVIDIA C1060 time gains over CPU and apeNEXT.

	high mass			low mass		
spatial size	32	48	64	16	32	48
Opteron (single core)	115	130	140	65	75	140
Xeon (single core)	85	85	100	30	40	50
apeNEXT crate	$\sim$6			$\sim$2		

Table 4: NVIDIA C2050 time gains over CPU and apeNEXT (same code as for C1060, no specific C2050 improvement implemented).

voted almost exclusively to the Dirac matrix inversion problem. We have shown that it is possible to use GPUs to efficiently perform a complete simulation, without the need to rely on more traditional architectures.

9. Acknowledgment

It's a pleasure to thank Edmondo Orlotti (NVIDIA) and Massimo Bernaschi (IAC, Rome) for the possibility they gave us to test our code on a C2050 card.

References

[1] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi, K. K. Szabo Comput. Phys. Commun. **177**, 631 (2007) [arXiv:hep-lat/0611022].

[2] NVIDIA Corporation, http://developer.nvidia.com/object/gpucomputing.html

[3] I. Horvath, A. D. Kennedy, S. Sint Nucl. Phys. B Proc. Suppl. **73**, 834 (1999) [arXiv:hep-lat/9809092].

[4] S. Duane, A. D. Kennedy, B. J. Pendleton, D. Roweth Phys. Lett. B **195**, 216 (1987).

[5] D. Göddeke, R. Strzodka, S. Turek International Journal of Parallel, Emergent and Distributed Systems **22**, 221 (2007).

[6] A. Greenbaum SIAM J. Matrix Anal. Appl. **18**, 535 (1997).

[7] H. K. van der Vorst, Q. Ye SIAM J. Sci. Comput. **22**, 835 (2000).

[8] M. A. Clark, R. Babich, K. Barros, R. C. Brower, C. Rebbi Comput. Phys. Commun. **181**, 1517 (2010) [arXiv:0911.3191 [hep-lat]].

[9] B. Jegerlehner [arXiv:hep-lat/9612014].